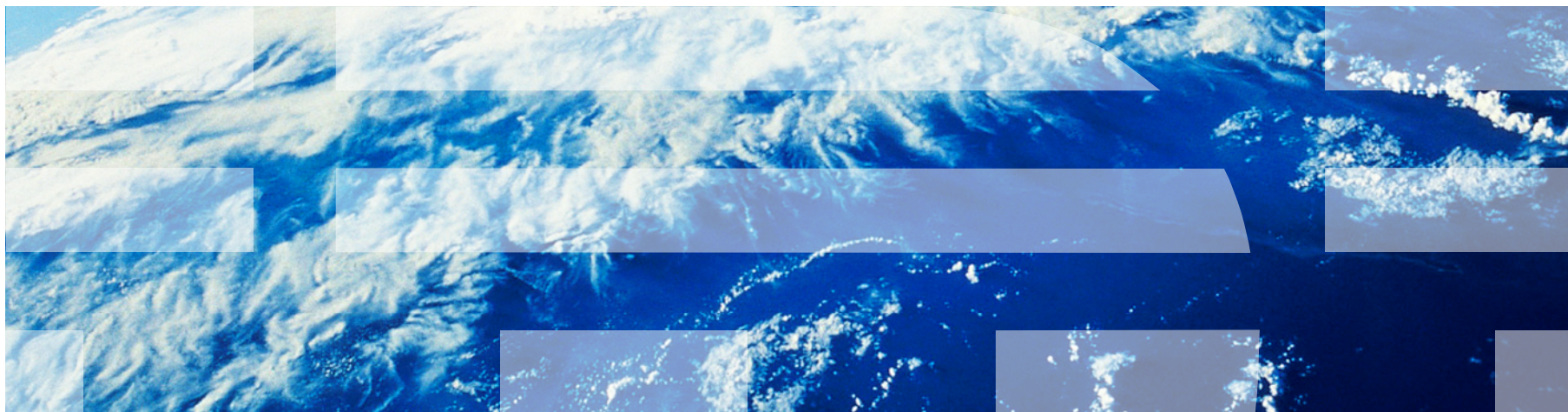

Lecture 9



Recap of lecture 8

■ Routing of queries and replication

- Master node architecture
 - Two alternative designs:
 1. All queries go through one master node
 2. Clients go directly to data nodes, occasionally sync up with master node
 - Advantage: simple
 - Disadvantage: single point of failure
- Distributed routing using consistent hashing
 - Advantage: no single point of failure, scalable
 - Disadvantage: worse performance (requires more communication)
- Replication
 - Master replica replicates to secondary nodes
 - Need to maintain consistency

■ HDFS

- Uses master node architecture
- Master node is called NameNode, data is stored on DataNodes
- Large immutable files
- Serves as file system for many distributed databases and analytics systems, including Spark/MapReduce

Recap of lecture 8 (continued)

■ Distributed Transactions and 2PC

- Each node has its own transaction coordinator and manager
- Any node can originate a transaction via coordinator
- 2PC has two phases:
 - Prepare phase: coordinator sends a prepare message to all managers, waits for ready response
 - Decision phase: if it received a ready response from all the managers, it can commit transaction locally and send commit message to all other nodes
- 2PC ensures that transactions are either committed or aborted on all nodes
- If any manager responds “not ready”, transaction must be aborted
- Coordinator can only continue if gets “ready” from all participating nodes
- 2PC can **block** if participating nodes have a “ready” log, but no commit/abort
 - Need to wait to hear from coordinator
- How to solve blocking?
 - Use **distributed consensus**, so that there are multiple coordinators, with a majority vote required

Today: MapReduce, Spark and Streaming

- MapReduce
 - Analytics programming interface
 - Word count example
 - Chaining
 - Reading/write from/to HDFS
 - Dealing with failures
- Spark
 - Motivation
 - Resilient Distributed Datasets (RDDs)
 - Programming interface
 - Transformations and actions
- Stream processing
 - Motivation
 - Spark streaming
 - Programming model
 - Twitter example

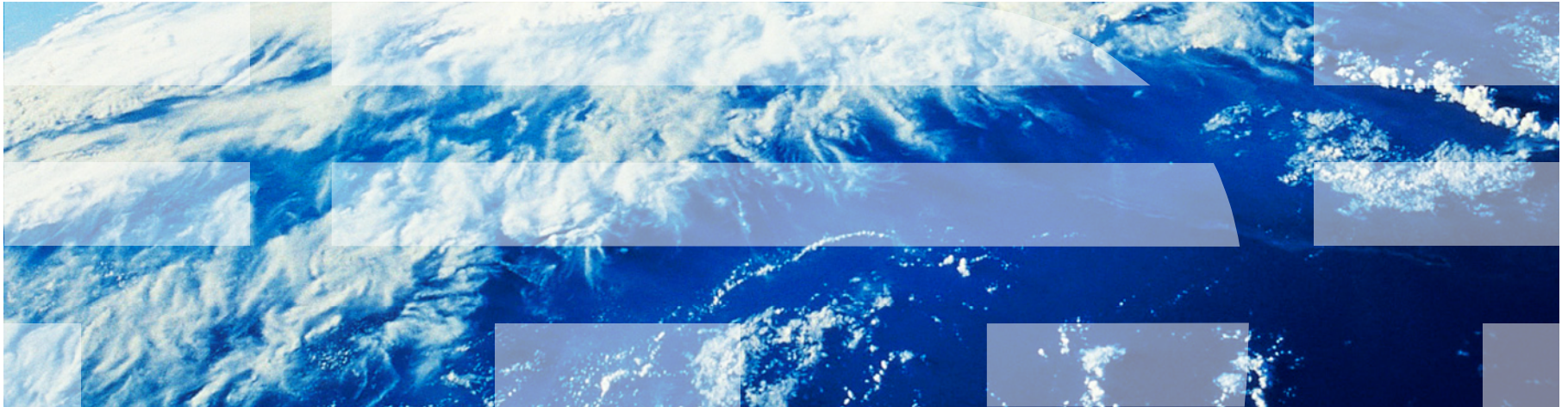
Logistics

- Almost done grading midterm
 - Should be done very soon, will announce it's done
 - Class is pass/fail, so grades are all about personalized feedback for each one of you on how well you know the material
- Syllabus
 - Classes will run until and including May 4
 - Instead of doing in-class midterm and final, we will be teaching
 - Will cover all material as planned originally, even though two classes were cancelled
 - Syllabus and materials are regularly updated on course website
- Final
 - Similar format to midterm
 - Probably: 1 week window, starting after the last class
 - Will include **all** material of the class, with an emphasis on material taught in the second half

Computer Systems for Data Science

Topic 6

MapReduce
Spark
Streaming



Borrowed from Jeff Ullman, Cristiana Amza and Indranil Gupta

MapReduce



MapReduce and Hadoop

- SQL and ACID are a very useful set of abstractions
- But: unnecessarily heavy for many tasks, hard to scale
- MapReduce is a more limited style of programming designed for:
 1. Easy parallel programming
 2. Invisible management of hardware and software failures
 3. Easy management of very-large-scale data
- It has several implementations, including Hadoop, Flink, and the original Google implementation just called “MapReduce.”
- It is also used in Spark

MapReduce in a Nutshell

- A MapReduce job starts with a collection of input elements of a single type.
 - Technically, all types are key-value pairs.
- Apply a user-written **Map function** to each input element, in parallel.
 - **Mapper** applies the Map function to a single element.
 - Many mappers grouped in a **Map task** (the unit of parallelism).
 - Usually a single Map task is run on a single node/server
- The output of the Map function is a set of 0, 1, or more *key-value pairs*.
- The system sorts all the key-value pairs by key, forming key-(list of values) pairs.

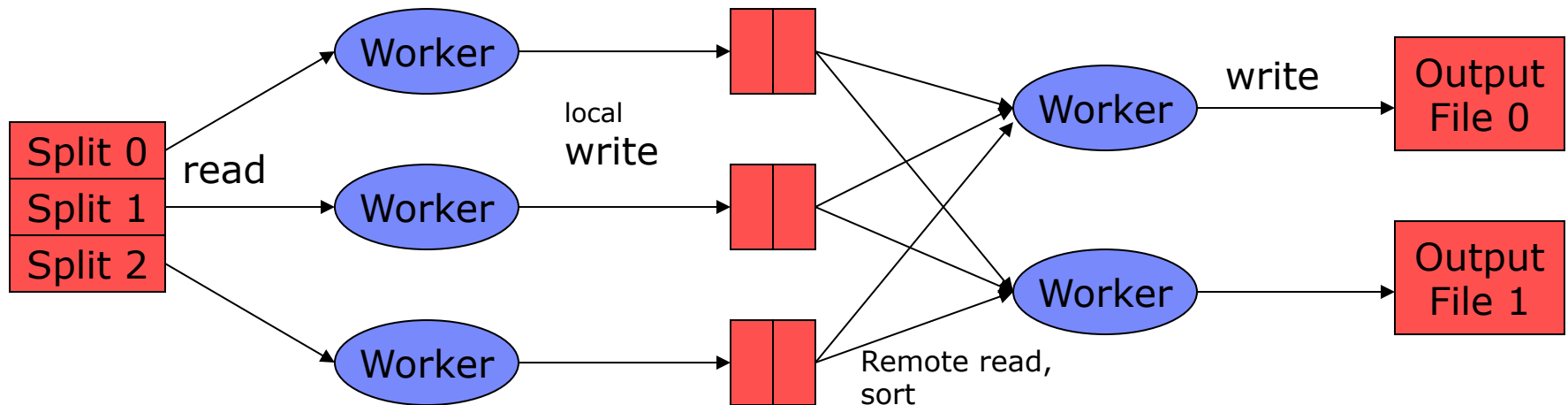
In a Nutshell – (2)

- Another user-written function, the **Reduce function**, is applied to each key-(list of values).
 - Application of the Reduce function to one key and its list of values is a **reducer**.
 - Often, many reducers are grouped into a **Reduce task**.
- Each reducer produces some output, and the output of the entire job is the union of what is produced by each reducer.

MapReduce workflow

Input Data

Output Data



Map

extract something you
care about from each
record

Reduce

aggregate,
summarize, filter,
or transform

Example: Word Count

- We have a large file of documents (the input elements).
- Documents are words separated by whitespace.
- Count the number of times each distinct word appears in the file.

Word Count Using MapReduce

```
map(key, value):
```

```
// key: document ID; value: text of document
```

```
  FOR (each word w IN value)
```

```
    emit(w, 1);
```

```
reduce(key, value-list):
```

```
// key: a word; value-list: a list of integers
```


```
  result = 0;
```

```
  FOR (each integer v on value-list)
```

```
    result += v;
```

```
  emit(key, result);
```

Expect to be all 1's, but
"combiners" allow local
summing of
integers with the same
key before passing to
reducers.



Mapper

- Reads in input pair `<Key, Value>`
- Outputs a pair `<K', V'>`
 - Let's count number of each word in user queries (or Tweets/Blogs)
 - The input to the mapper will be `<queryID, QueryText>`:
`<Q1, "The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am." >`
 - The output would be:
`<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>`

Reducer

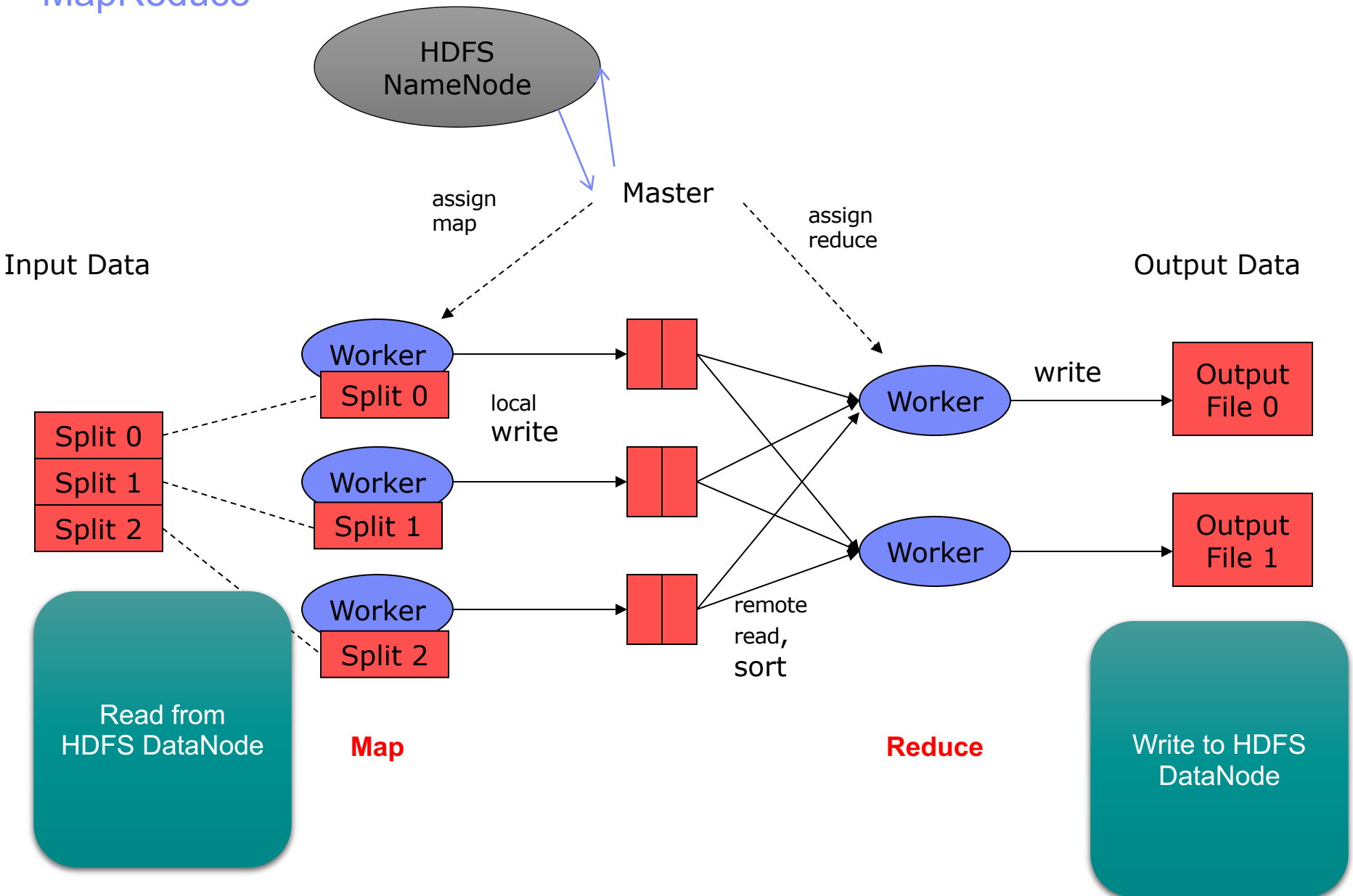
- Accepts the Mapper output, and aggregates values on the key
 - For our example, the reducer input would be:
<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <**store**, 1> <the, 1> <store, 1>
<was, 1> <closed, 1> <the, 1> <**store**, 1> <opens,1> <in, 1> <the, 1> <morning, 1>
<the 1> <**store**, 1> <opens, 1> <at, 1> <9am, 1>
 - The output would be:
<The, 6> <teacher, 1> <went, 1> <to, 1> <**store**, 3> <was, 1> <closed, 1> <opens,
1> <morning, 1> <at, 1> <9am, 1>

Another example: Chaining MapReduce

Count of URL access frequency

- Input: Log of accessed URLs, e.g., from proxy server
- Output: For each URL, % of total accesses for that URL
- First step:
 - Map – process web log and outputs $\langle \text{URL}, 1 \rangle$
 - Multiple Reducers - Emits $\langle \text{URL}, \text{URL_count} \rangle$
(So far, like Wordcount. But still need %)
- Chain another MapReduce job after above one
 - Map – processes $\langle \text{URL}, \text{URL_count} \rangle$ and outputs $\langle 1, (\langle \text{URL}, \text{URL_count} \rangle) \rangle$
 - 1 Reducer – Does two passes. In first pass, sums up all URL_count 's to calculate overall_count . In second pass calculates %'s
Emits multiple $\langle \text{URL}, \text{URL_count}/\text{overall_count} \rangle$

MapReduce



Locality Optimization

- **Master scheduling policy:**
 - Asks HDFS for locations of replicas of input file blocks
 - Map tasks scheduled so HDFS input block replica are on same machine or same rack
- **Effect:** Thousands of machines read input at local disk speed
 - Don't need to transfer input data all over the cluster over the network: eliminate network bottleneck!

Failure in MapReduce

- Failures are the norm in data centers
- **Worker failure**
 - Master detects if workers failed by periodically pinging them (this is called “heartbeat”)
 - Re-execute in-progress map/reduce tasks
- **Master failure**
 - Single point of failure; Resume from Execution Log
- **Robust**
 - Google’s experience: lost 1600 of 1800 machines once, but finished fine

Refinement: Redundant Execution

- Slow workers or **stragglers** significantly lengthen completion time
 - Slowest worker can determine the total latency!
 - This is why many systems measure **99th percentile latency**
 - Other jobs consuming resources on machine
 - Bad disks with errors transfer data very slowly
- **Solution:** spawn backup copies of tasks
 - Whichever one finishes first "wins"
 - I.e., treat slow executions as failed execute

Borrowed from Indranil Gupta, Faria Kalim, Patrick Wendell

Spark



Motivation

- Map reduce based tasks are slow
 - Data written to and read from storage
 - In the beginning and end of each Map and Reduce task
- Iterative algorithms not supported
 - Need to chain map reduce jobs → cumbersome, need to know how many jobs in advance (hard to do a loop)
- No support for interactive data mining and analytics

Spark's Key Concept: Resilient Distributed Datasets (RDDs)

- A form of **distributed shared memory**
 - Eliminates the need to read/write to/from disk intermediate data between iterations
 - Read only / immutable, partitioned collections of records in memory
 - Deterministic
 - Formed by specific operations (map, filter, join, etc.)
 - Can be read from stable storage or other RDDs
- More expressive interface than MapReduce
 - Transformations (e.g. map, filter, groupBy)
 - Actions (e.g. count, collect, save)
- Recent versions of Spark introduced Datasets/Dataframes
 - Like an RDD, but you can run SQL queries over it
 - Organized into rows columns, similar to DB relation

Spark programming interface

- Lazy operations
 - Transformations not done until action
- Operations on RDDs
 - Transformations - build new RDDs
 - Can include both traditional map and/or reduce operations
 - Actions - compute and output results
 - E.g., to a file, to a Python collection
- Partitioning – layout across nodes
- Persistence – final output can be stored on disk

RDD on Spark

Transformations	$map(f : T \Rightarrow U)$: $RDD[T] \Rightarrow RDD[U]$ $filter(f : T \Rightarrow Bool)$: $RDD[T] \Rightarrow RDD[T]$ $flatMap(f : T \Rightarrow Seq[U])$: $RDD[T] \Rightarrow RDD[U]$ $sample(fraction : Float)$: $RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling) $groupByKey()$: $RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$ $reduceByKey(f : (V, V) \Rightarrow V)$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $union()$: $(RDD[T], RDD[T]) \Rightarrow RDD[T]$ $join()$: $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$ $cogroup()$: $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$ $crossProduct()$: $(RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$ $mapValues(f : V \Rightarrow W)$: $RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning) $sort(c : Comparator[K])$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $partitionBy(p : Partitioner[K])$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
Actions	$count()$: $RDD[T] \Rightarrow Long$ $collect()$: $RDD[T] \Rightarrow Seq[T]$ $reduce(f : (T, T) \Rightarrow T)$: $RDD[T] \Rightarrow T$ $lookup(k : K)$: $RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs) $save(path : String)$: Outputs RDD to a storage system, <i>e.g.</i> , HDFS

Table 2: Transformations and actions available on RDDs in Spark. $Seq[T]$ denotes a sequence of elements of type T .

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

Transformed RDD

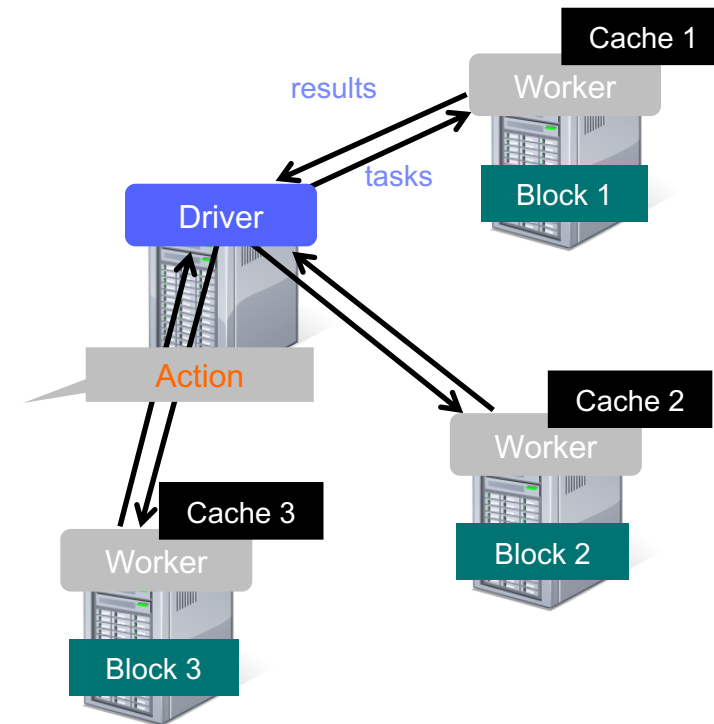
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

...

Full-text search of Wikipedia

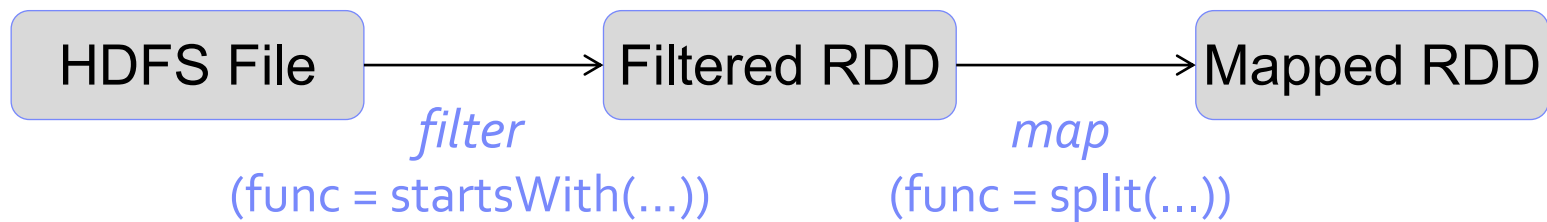
- 60GB on 20 EC2 machine
- 0.5 sec vs. 20s for on-disk



Fault Recovery

RDDs track **lineage** information that can be used to efficiently recompute lost data

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))  
               .map(lambda s: s.split("\t")[2])
```



Creating RDDs

Turn a Python collection into an RDD

```
> sc.parallelize([1, 2, 3])
```

Load text file from local FS, HDFS, or S3

```
> sc.textFile("file.txt")
```

```
> sc.textFile("directory/*.txt")
```

```
> sc.textFile("hdfs://namenode:9000/path/file")
```

Basic Transformations

```
> nums = sc.parallelize([1, 2, 3])
```

```
# Pass each element through a function
```

```
> squares = nums.map(lambda x: x*x) // {1, 4, 9}
```

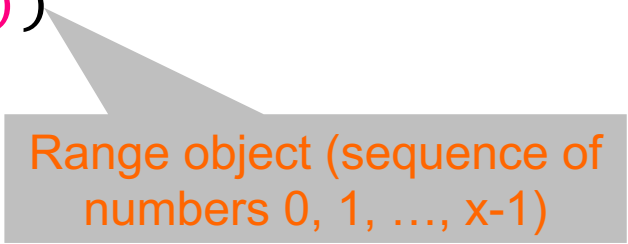
```
# Keep elements passing a predicate
```

```
> even = squares.filter(lambda x: x % 2 == 0) // {4}
```

```
# Map each element to zero or more others
```

```
> nums.flatMap(lambda x: => range(x))
```

```
> # => {0, 0, 1, 0, 1, 2}
```



Range object (sequence of numbers 0, 1, ..., x-1)

Basic Actions

```
> nums = sc.parallelize([1, 2, 3])  
  
# Retrieve RDD contents as a local collection  
> nums.collect() # => [1, 2, 3]  
  
# Return first K elements  
> nums.take(2)    # => [1, 2]  
  
# Count number of elements  
> nums.count()    # => 3  
  
# Merge elements with an associative function  
> nums.reduce(lambda x, y: x + y) # => 6  
  
# Write elements to a text file  
> nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

Spark's “distributed reduce” transformations operate on RDDs of key-value pairs

Python:

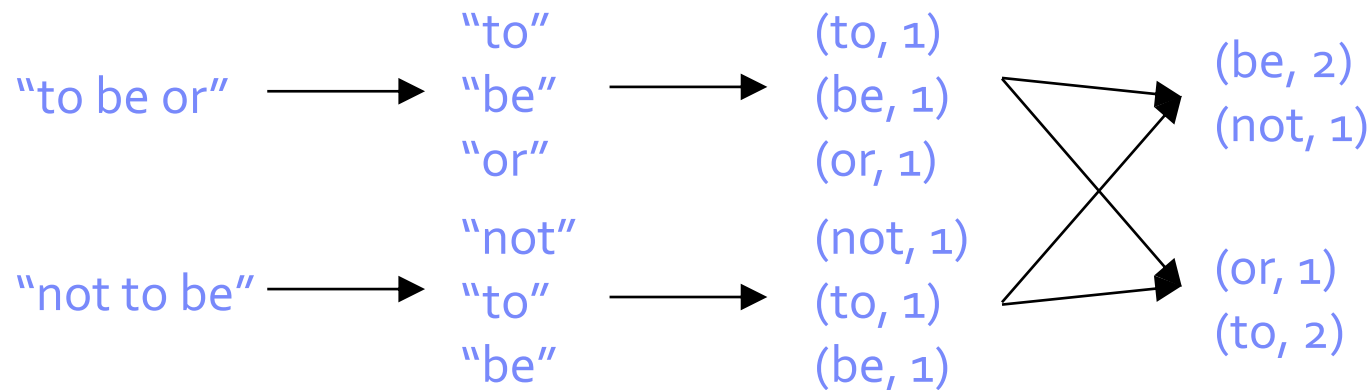
```
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

Some Key-Value Operations

```
> pets = sc.parallelize(  
    [("cat", 1), ("dog", 1), ("cat", 2)])  
> pets.reduceByKey(lambda x, y: x + y)  
    # => {(cat, 3), (dog, 1)}  
> pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}  
> pets.sortByKey()  # => {(cat, 1), (cat, 2), (dog, 1)}
```


Example: Word Count

```
> lines = sc.textFile("hamlet.txt")  
> counts = lines.flatMap(lambda line: line.split(" "))  
                    .map(lambda word => (word, 1))  
                    .reduceByKey(lambda x, y: x + y)
```

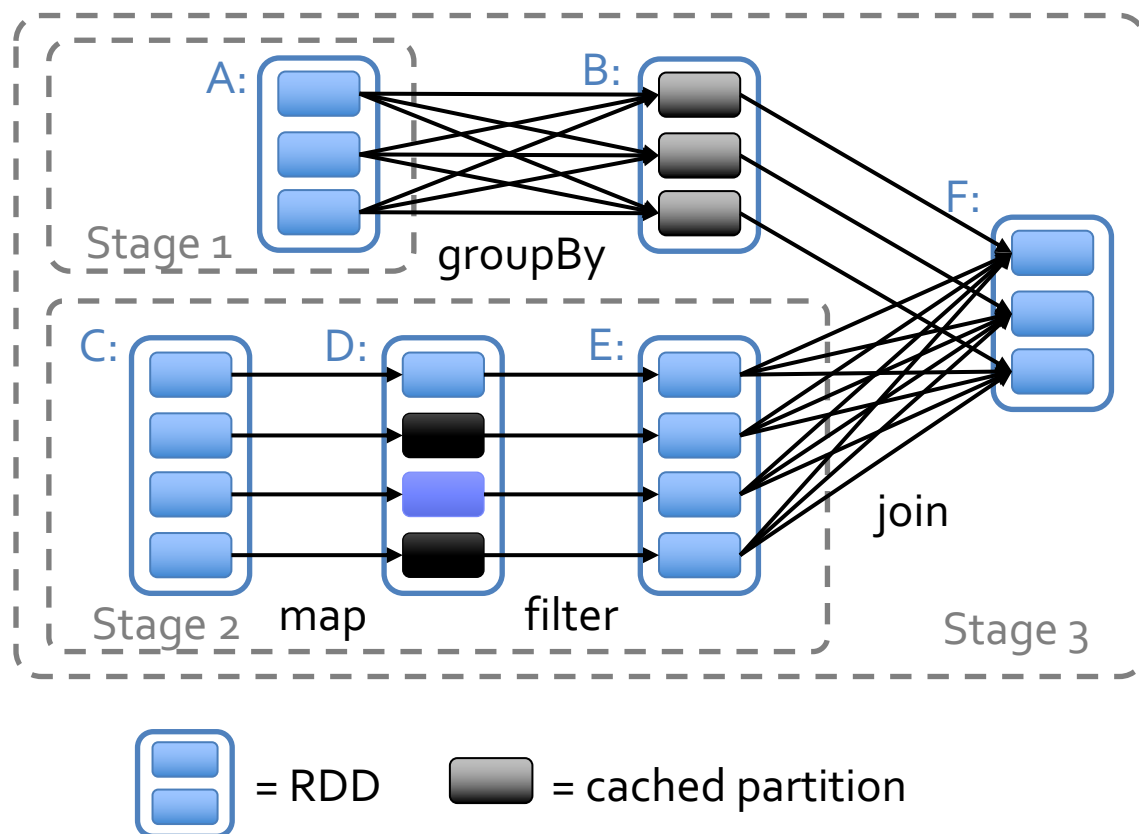


More RDD Operators

- | | | |
|------------------|---------------|-------------|
| ■ map | ■ reduce | sample |
| ■ filter | ■ count | take |
| ■ groupBy | ■ fold | first |
| ■ sort | ■ reduceByKey | partitionBy |
| ■ union | ■ groupByKey | mapWith |
| ■ join | ■ cogroup | pipe |
| ■ leftOuterJoin | ■ cross | save |
| ■ rightOuterJoin | ■ zip | ... |

Under The Hood: DAG Scheduler

- General task graphs
- Automatically pipelines functions
- Data locality aware
- Partitioning aware to avoid shuffles



Lecture 10



Recap of lecture 9

■ **MapReduce**

- A programming interface that is tailored for distributed computation
 - Gave up on SQL
- Map takes an input file and maps it to key, value pairs
- Reduce aggregates on all the values of a particular key
- MapReduce can be chained for more complicated queries
- Requires reading/writing from/to storage at each step
 - Tightly integrated with Hadoop File System
- Limitations: storage slows every step down, not good for "foor-loops" or complicated iterative queries

Recap of lecture 9 (continued)

■ Spark

- Much more expressive programming interface than MapReduce
 - Supports Python/SQL/R/...
- Uses memory for all intermediate steps
- Memory is stored in RDDs (and these days in Dataframes and Datasets)
- RDDs are read-only
- Basic idea: apply transformations on RDDs, and an action as the final output
 - Transformations are deterministic, and can be replayed
- Uses a DAG (Directed Acyclic Graph) to schedule tasks and tracks lineage to re-compute data on failed nodes
- Does not replicate RDDs

Today: UDF and Regex, Streaming, Caching

- Spark (continued)
 - UDF (User Defined Functions)
 - Regular expressions (Regex)
- Stream processing
 - Motivation
 - Spark streaming
 - Programming model
 - Twitter example
- Caching
 - FIFO
 - LRU
 - LFU
 - ...

Logistics

- Midterm grading is done
- More resources for Spark homework
 - Hongyi posted some more resources on Piazza
 - We will now discuss some extra Spark topics that may be useful for the homework assignment
- Please conserve credits
 - This is good practice for the course and for life!
 - It's inconvenient, but shut down your VMs after you are done working

SQL on Spark

- Spark SQL allows you to use SQL on Spark
- Instead of using RDDs, it uses DataFrames
 - Like an RDD, but in a table format
 - Each column has a name
- A few useful operations:
- `df.collect()`
 - `[Row(price=100, company='Ford'), Row(price=5, company='VW')]`
- `df.columns`
 - `['price', 'company']`
- `df.count()`
 - `2`
- `df.filter(df.price > 50).collect()`
 - `[Row(price=100, company='Ford')]`

Return records as a list

Returns column format

Returns number of rows

Filters

User-Defined Functions (UDF) in Spark

- Sometimes you need to write a custom operation that is run on each row
 - Cleaning/mapping/filtering strings
 - Custom math operation
- For example:
 - For a column that contains names, remove all middle names and just keep first and last name
 - Extract the name of the service from an error log
 - Change the names “Nick, Rick” to “Nicholas, Richard”
- UDF allows you to define a new custom function that will be applied to each row
 - Word of caution: be sure to consider NULL case
 - Most commonly within the UDF itself
 - Need to register UDF function (different name than the Python function)

UDF Example

```
from pyspark.sql.types import StringType
from pyspark.sql.functions import udf, col

def clean_middle(name):
    split_name = name.split()
    if (len(split_name) > 2):
        return split_name[0] + " " + split_name[-1]
    else:
        return name

clean_middle_udf = udf(lambda name: clean_middle(name), StringType())
names = [{"name": "Jane Smith"}, {"name": "John J Smith"}]
df = spark.createDataFrame(names)
df.withColumn("no_middle", clean_middle_udf(col("name"))).show()
```

```
+-----+-----+
|      name| no_middle|
+-----+-----+
| Jane Smith| Jane Smith|
| John J Smith| John Smith|
+-----+-----+
```

UDF Example (raw text)

```
from pyspark.sql.types import StringType
from pyspark.sql.functions import udf, col

def clean_middle(name):
    split_name = name.split()
    if (len(split_name) > 2):
        return split_name[0] + " " + split_name[-1]
    else:
        return name

clean_middle_udf = udf(lambda name: clean_middle(name), StringType())

names = [{"name": "Jane Smith"}, {"name": "John J Smith"}]
df = spark.createDataFrame(names)
df.withColumn("no_middle", clean_middle_udf(col("name"))).show()
```

Regular expressions (Regex)

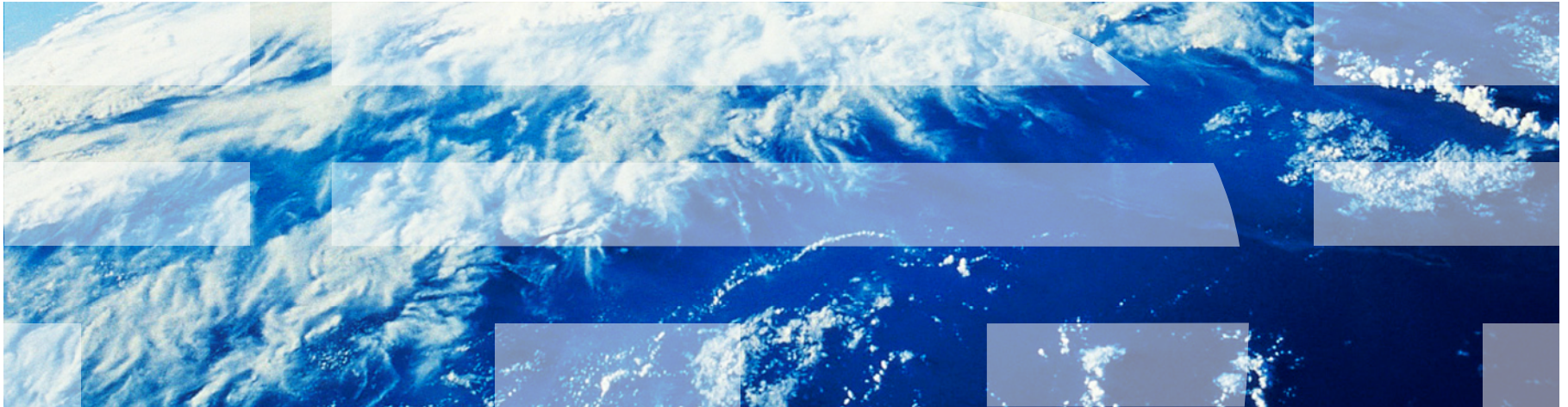
- A scripting language for matching and/or manipulating strings of text
 - Manipulating text with built-in Python string functions can be cumbersome
 - Python and PySpark support regex
- Examples of operations
 - Find all substrings with “helloworld” and replace with: “Hello World”
 - Find all substrings with “fox”, but exclude “foxtrot”, and replace with the word “wolf”
 - Remove all special characters (e.g., !%&), punctuation (e.g., .,:;)
- Useful in many programming contexts
 - Spark
 - Python
 - SQL
 - Etc.
- Hint: can be useful in homework assignment (coupled with a UDF perhaps?)
- Use free online regex tool for debugging: <https://regex101.com/>

Regex matching examples

- Text: fox foxtrot ox box
- fox: fox foxtrot ox box
- ox: fox foxtrot ox box
- fox\w: fox foxtrot ox box (\w matches any word character)
- fox\w*: fox foxtrot ox box (* matches zero or more of the preceding character)
- fox\w+: fox foxtrot ox box (+ matches one or more of the preceding character)
- [^f]ox: fox foxtrot ox box (without the letter f in the beginning)
- [^f\W]ox: fox foxtrot ox box (without the letter f or a non-word character)
- [bf]ox: fox foxtrot ox box (match on b or f followed by ox)

Borrowed from Tathagata Das and Indranil Gupta

Stream Processing



Motivation

- Large amounts of data => Need for real-time views of data
 - Social network trends, e.g., Twitter real-time search
 - Website statistics, e.g., Google Analytics
 - Intrusion detection systems, e.g., in most datacenters
- Process large amounts of data
 - With latencies of few seconds
 - With high throughput

Would MapReduce or normal Spark work?

- **Batch Processing** => need to wait for entire computation on large dataset to complete
- Not intended for long-running and real-time stream-processing

Which one of these is a stream processing job?

- A) Uber
Calculating surge prices
- B) LinkedIn
Aggregating updates into one email
- C) Netflix
Understanding user behavior to improve personalization
- D) TripAdvisor
Calculating earnings per day & fraud detection
- E) All of them

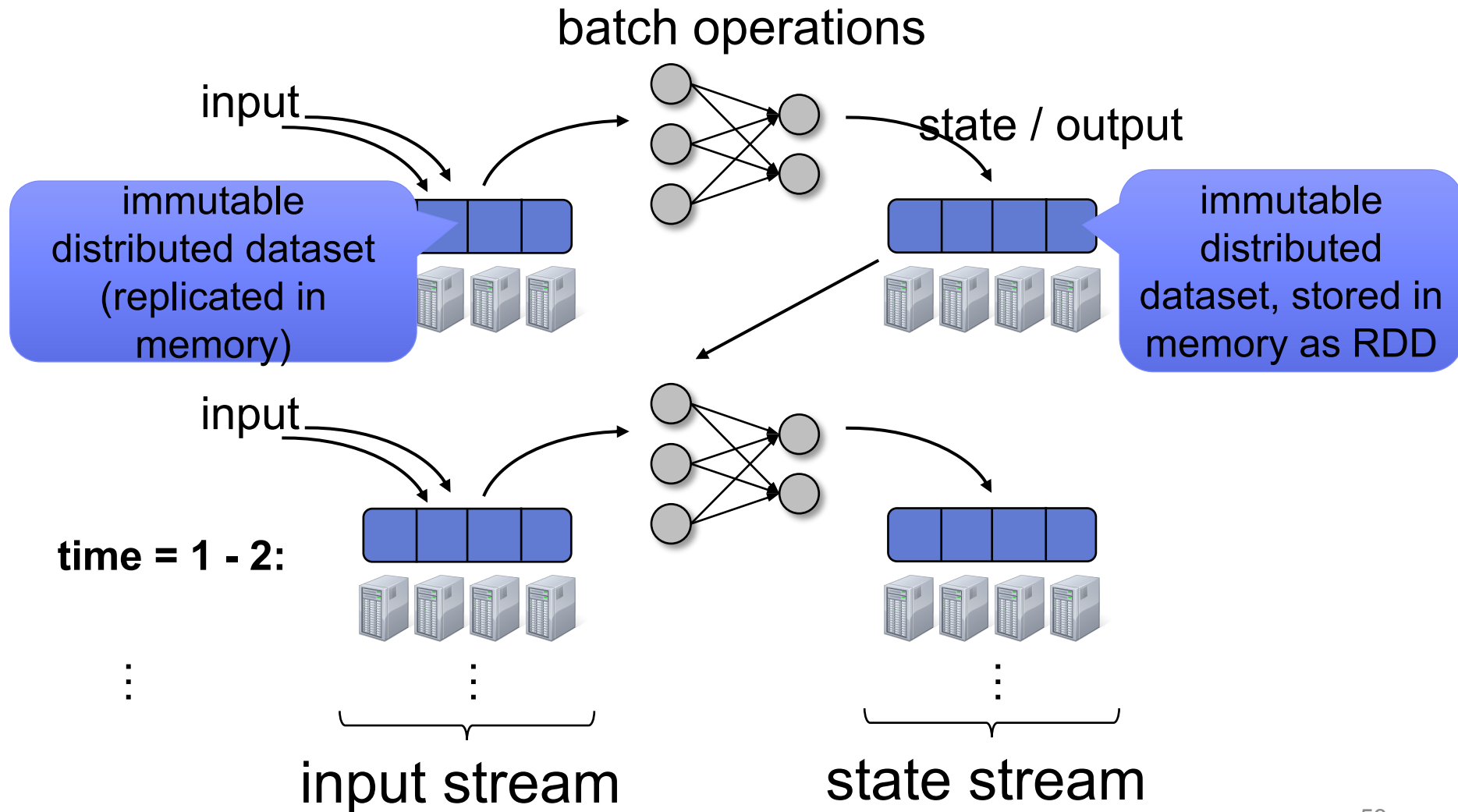
Discretized Stream Processing

- Run a streaming computation as a **series of very small, deterministic batch jobs**
- Batch processing models, like MapReduce, recover from faults and stragglers efficiently
 - Divide job into deterministic tasks
 - Rerun failed/slow tasks in parallel on other nodes
- Same recovery techniques at lower time scales
 - Transformations are not lost (or performed twice) if a worker dies

Spark Streaming

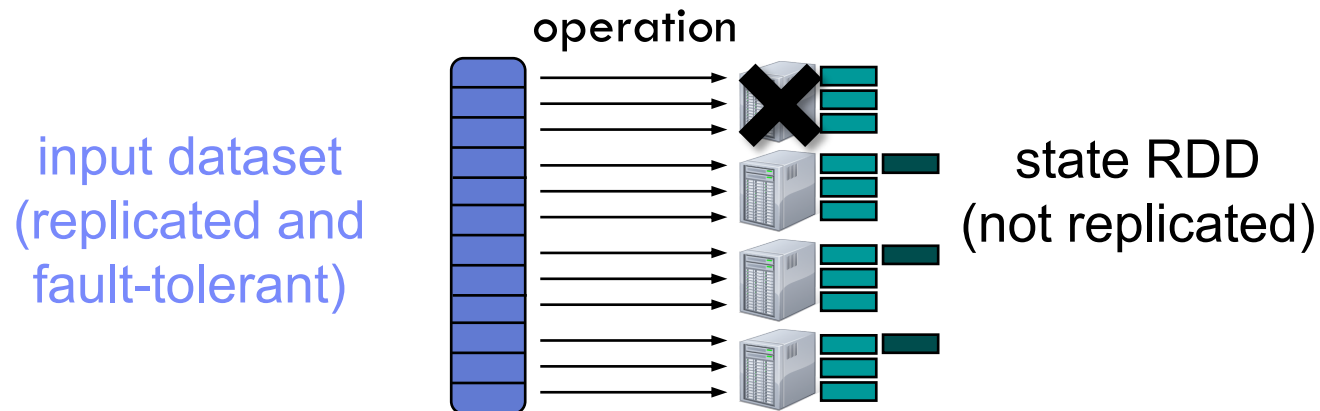
- State between batches kept in memory as **immutable, fault-tolerant dataset**
 - Specifically as an RDD/Dataframe/Dataset
- Batch sizes can be reduced to as low as 1/2 second to achieve ~ 1 second latency
- Combines streaming and batch workloads
- Many other alternatives:
 - Apache Storm
 - Apache Flink
 - Amazon Kinesis
 - Google Dataflow
 - ...

Discretized Stream Processing



Fault Recovery

- State stored as RDD
 - Deterministically re-computable parallel collection
 - Remembers lineage of operations used to create them
- Fault / straggler recovery is done **in parallel** on other nodes



Fast recovery from faults without
full data replication

Programming Model

- A Discretized Stream or **DStream** is a series of RDDs representing a stream of data
 - API *very similar* to RDDs
- DStreams can be created...
 - Either from live streaming data
 - Or by transforming other DStreams

DStream Data Sources

- Many data sources can be inputs
 - HDFS
 - Kafka
 - Flume
 - Twitter
 - ...

Transformations

Build new streams from existing streams

- Filters/aggregate operations
 - map, flatMap, filter, count, reduce,
 - groupByKey, reduceByKey, sortByKey, join
 - etc.
- New window and stateful operations
 - window, countByWindow, reduceByWindow
 - countByValueAndWindow, reduceByKeyAndWindow
 - updateStateByKey
 - etc.

Output Operations

Send data to outside world

- `saveAsHadoopFiles`
- `print` – prints on the driver's screen
- `foreach` - arbitrary operation on every RDD

Example

Process a stream of Tweets to find the 20 most popular hashtags in the last 10 mins

1. Get the stream of Tweets and isolate the hashtags
2. Count the hashtags over 10 minute window

1. Get the stream of Hashtags

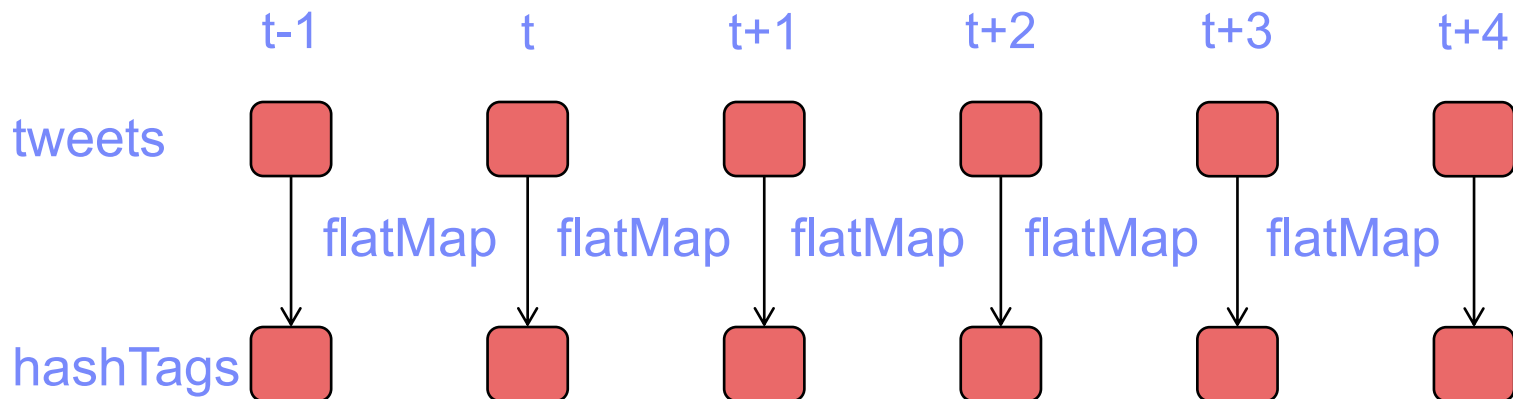
```
tweets = ssc.twitterStream(username, password)
```

DStream

```
hashtags = tweets.flatMap (lambda tweet: tweet.getTags())
```

transformation

 = RDD

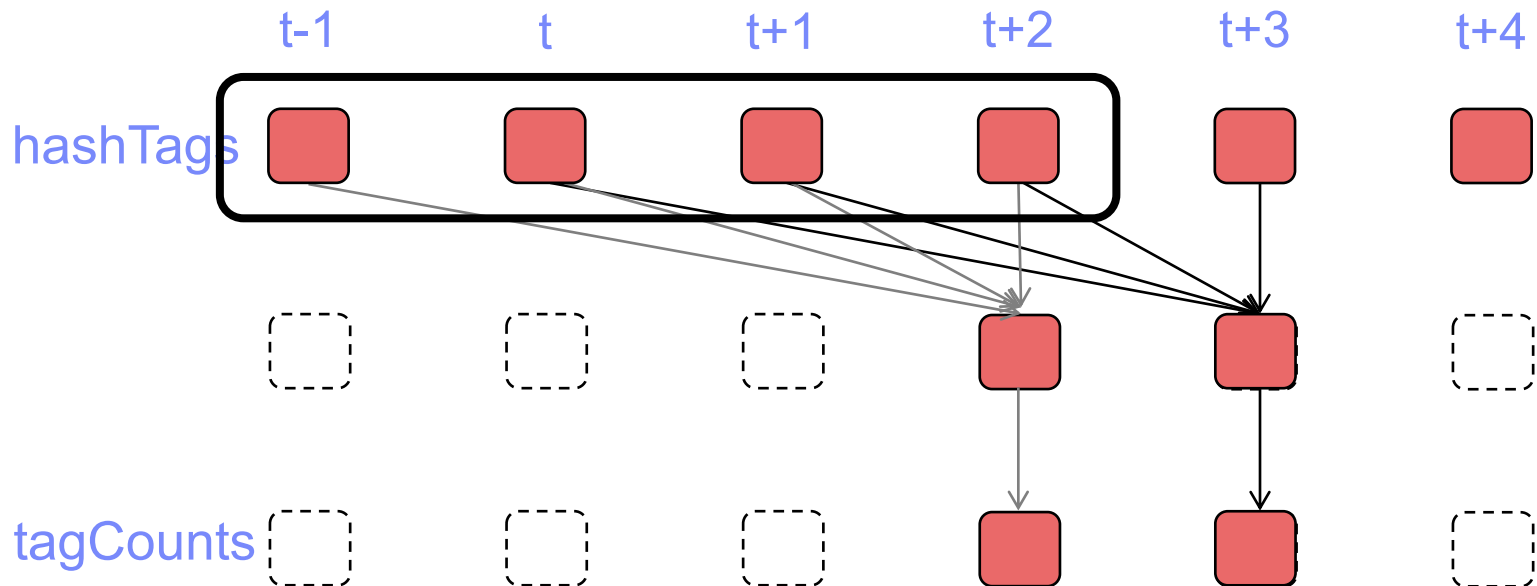


2. Count the hashtags over 10 min

```
tweets = ssc.twitterStream(userName)
hashtags = tweets.flatMap (lambda tweet: tweet.getTags())
tagCounts = hashtags.window(Minutes(10), Seconds(1)).countByValue()
```

Window
length

Sliding
interval



2. Count the hashtags over 10 min

```
tweets = ssc.twitterStream(username, password)
```

```
hashtags = tweets.flatMap (lambda tweet: tweet.getTags())
```

```
tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```

